iTransformer: Using SSD to Improve Disk Scheduling for High-Performance I/O

Kei Davis, CCS-7; Song Jiang, Xuechen Zhang, Wayne State University

As part of an ongoing collaboration the authors worked together at LANL during the summer of 2011 under the DOE Faculty and Student Team (FaST) program. Here we briefly summarize one of the summer's efforts, notably enabled by the availability of the CCS-7 experimental cluster Darwin. This work contributed to CCS-7 research into common runtime elements for programming models for increasingly parallel scientific applications and computing platforms [1].

Data-intensive scientific computing applications are producing increasingly high Input/Output (I/O) demands on the storage devices of high-performance computing systems. Request concurrency, or the number of processes concurrently issuing requests, can be very high at data servers that are serving requests from applications running on a large-scale cluster. Besides the potentially large volume of requested data, this concurrency can significantly compromise the efficiency of a hard-disk-based storage system: data on a disk that are requested by different processes or programs are usually spatially separated on the disk, and concurrently accessing them can cause the disk heads to frequently seek from track to track, potentially delivering an I/O throughput an order of magnitude lower (or worse) than that for sequential disk access.

The emerging solid state drive (SSD) is largely unaffected by random access because it is, effectively, a uniform memory access device. However, it is currently not economical in high-performance computing for use as the main storage in a large-scale installation. More cost-effective and practical options are either to use an SSD as buffer cache between main (dynamic random-access memory, DRAM) memory and the hard disk and exploit workloads' locality for data caching, or use SSD with a hard disk to form a hybrid storage device such that frequently accessed data is stored on the SSD.

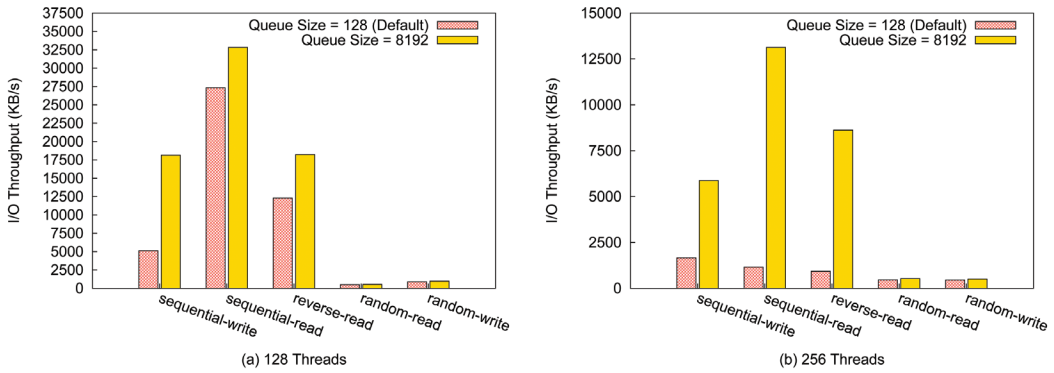

Fig. 1. I/O throughput of a data server running IOzone with (a) 128 threads; and, (b) 256 threads. In each figure I/O throughputs with differing dispatch queue sizes and differing data access patterns are shown.

These schemes for SSD usage, however, do not effectively address the problem of concurrent requests from data-intensive parallel programs. First, the data accessed in a single run of such a program can be larger than the capacity of the SSD. When a program processes a large data set, data are rarely accessed multiple times from the disk, and the accesses therefore exhibit weak temporal locality, which is hard to exploit for effective caching by a relatively small SSD. Second, requests to a disk are usually interleaved from different processes of one or multiple programs. Most existing SSD-based schemes exploit spatial locality, that is, they attempt placement of randomly accessed data on the SSD such that the hard disk serves requests of sequential, or at least well-ordered, data. However, when the request concurrency is high, it is highly likely that most requests from different processes will be presented to the disk as random access and will need to be handled by the SSD. This would overwhelm the SSD as a cache, or as a small storage device for random data, and make these schemes ineffective.

In the operating system, the I/O scheduler is the last opportunity to exploit spatial locality in the presence of high request concurrency. For example, CFQ, the default Linux disk scheduler, reduces random data accesses by merging and sorting outstanding requests according to their logical block addresses (LBA). Outstanding requests are kept in a data structure called a dispatch queue. The larger the queue, the more requests can be collected for sorting and the greater the chance to

For more information contact Kei Davis at Kei@lanl.gov

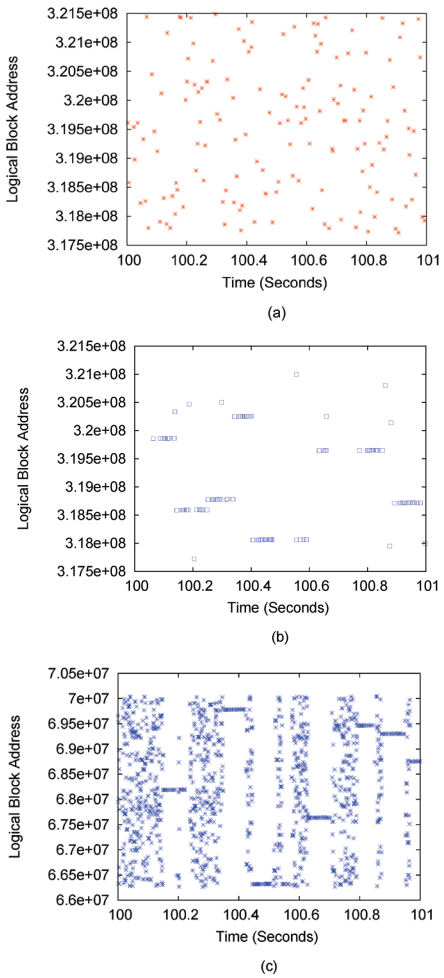

Fig. 2. Accessed locations when running ior-mpi-io using read requests in a sampled one-second execution period. (a) The locations are on the hard disk and the stock system is used. (b) The locations are on the hard disk and the system with iTransformer. (c) The locations are on the SSD and the system with iTransformer.

exploit spatial locality. The default queue depth in Linux's CFQ is 128.

We started by investigating the effect of simply increasing queue size on I/O performance in the presence of request concurrency by running IOzone, a commonly used file system benchmark, to generate a variety of file operations with varying queue size on a data server running Linux with CFQ to access data on a hard disk. Figure 1 shows I/O throughputs reported by the benchmark for access patterns Sequential Read/Write, Reverse Read, and Random Read/Write, with queue sizes 128 and 8192, with either 128 threads (Fig. 1a) or 256 threads (Fig. 1b).

This investigation showed that increasing queue size can significantly improve performance for Sequential Read/Write and Reverse Read. When the queue size is increased to 8192 the throughputs are significantly increased by 42% to 650%. This demonstrates that a large queue can effectively recover spatial locality if it exists in requests from the same thread. However, when individual threads issue fully random requests, the I/O throughputs are very low and the improvements made by the increased queue size are also small. This shows that random requests are at best difficult to schedule for efficient service by hard disk.

While increasing the size of the dispatch queue in memory can improve access locality for higher disk efficiency, by itself the approach has limitations. First, having a large queue would allow many write requests to be outstanding in volatile DRAM memory, running the risk of losing a large amount of data where frequent system failures are expected to be the norm. Second, although a long queue usually improves throughput, it can allow requests to remain in the queue for an extended period of time without being completed, which may result in excessive response times for those requests—for applications with strict quality of service (QoS) requirements a long queue can be problematic. Third, as we showed in the experiments, simply increasing the queue size may not be sufficient, especially for addressing the issue of concurrency among streams of random requests.

Our design started with extending the scheduler's dispatch queue using SSD to hold the extension. Various algorithmic techniques were devised to overcome all of the aforementioned limitations. Our implementation of this scheduling architecture and the scheduling algorithm, collectively called iTransformer, is as a stand-alone Linux kernel module. The implementation is transparent to the software above the generic block layer in the kernel memory hierarchy and is therefore portable across different parallel file systems. Our evaluation of iTransformer was performed on the CSS-7 Darwin Cluster with a suite of representative benchmarks. We used 48 processes per node (one process per core on the DL585 nodes), one SSD per node, and the PVFS2 parallel virtual file system. iTransformer significantly reduced random access of the hard disks and increased I/O throughput of the storage system by up to three times and 35% on average, as compared to the stock system, for the benchmarks, using a mere 8 GB of each SSD. Figure 2 gives an example of how accesses are serialized to hard disk.

[1] Zhang, X. et al., "iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O," *26th IEEE Int Parallel Distr Process Symp,* to appear (2012).

Funding Acknowledgments
DOE NNSA, Advanced Simulation and Computing, Computational Systems and Software Environments